Method Overriding vs Overloading

Let's review the **main differences between method overriding and method overloading**.

Method Overloading

Method overloading means providing two or more separate methods, in a class, with the **same name**, but **different parameters**.

Method return type may or may not be different, and that allows us to reuse the same method name.

Overloading is very handy, it **reduces duplicated code**, and we don't have to remember multiple method names.

We can overload static, or instance methods.

Method Overloading

To the code calling an overloaded method, it looks like a single method can be called, with different sets of arguments.

In actuality, each call that's made with a different set of arguments, is calling a separate method.

Java developers often refer to method overloading, as compile-time polymorphism.

This means the compiler is determining the right method to call, based on the method name and argument list.

Method Overloading

Usually **overloading** happens within a **single class**.

But methods can also be overloaded by subclasses.

That's because, a subclass inherits one version of the method from the parent class, and then the subclass can have another overloaded version of that method.

Method Overloading Rules

Methods will be considered overloaded if both methods follow the following rules:

- Methods must have the same method name.
- Methods must have different parameters.

If methods follow the rules above:

- They may or may not have different return types.
- They may or may not have different access modifiers.
- They may or may not throw different checked or unchecked exceptions.

Method Overriding

Method overriding, means defining a method in a child class, that already exists in the parent class, with the same signature (the **same name**, **same arguments**).

By extending the parent class, the child class gets all the methods defined in the parent class (those methods are also known as derived methods).

Method overriding is also known as **Runtime Polymorphism**, or **Dynamic Method Dispatch**, because the method that is going to be called, is decided at runtime, by the Java virtual machine.

Method Overriding

When we **override** a method, it's recommended to put **@Override**, immediately above the method definition.

The @Override statement is not required, but it's a way to get the compiler to flag an error, if you don't actually properly override this method.

We'll get an error, if we don't follow the overriding rules correctly.

We can't override static methods, **only instance methods** can be overridden.

Method Overriding Rules

A method will be considered overridden, if we follow these rules.

- It must have the same name and same arguments.
- The return type can be a subclass of the return type in the parent class.
- It can't have a lower access modifier. In other words, it can't have more restrictive access privileges.
- For example, if the parent's method is protected, then using private in the child's overridden method is not allowed. However, using public for the child's method would be allowed, in this example.

Method Overriding Rules

There's also some important points about method overriding to keep in mind.

- Only **inherited methods** can be overridden, in other words, methods can be overridden only in child classes.
- Constructors and private methods cannot be overridden.
- Methods that are final cannot be overridden.
- A subclass can use `super.methodName()` to call the superclass version of an overridden method.

Method Overriding vs. Overloading

OVERRIDING

same name
same parameters

```
class Dog {

    public void bark() {
        System.out.println("woof");
    }
}

class GermanShepherd extends Dog {

    @Override
    public void bark() {
        System.out.println("woof woof woof");
    }
}
```

OVERLOADING

same name
different parameters

```
class Dog {

    public void bark() {
        System.out.println("woof");
    }

    public void bark(int number) {
        for(int i = 0; i < number; i++) {
            System.out.println("woof");
        }
    }
}
```

Method Overriding vs. Overloading

Method Overloading	Method Overriding
Provides functionality to reuse a method name with different parameters.	Used to override a behavior which the class has inherited from the parent class.
Usually in a single class but may also be used in a child class.	**Always in two classes** that have a child-parent or IS-A relationship.
Must have different parameters.	**Must have** the same parameters and same name.
May have different return types.	**Must have** the same return type or covariant return type(child class).
May have different access modifiers(private, protected, public).	**Must NOT** have a lower modifier but may have a higher modifier.
May throw different exceptions.	**Must NOT** throw a new or broader checked exception.

Covariant Return Type

The return type of an overridden method can be the same type as the parent method's declaration.

But it can also be a subclass.

The term, covariant return type, is more appropriate.

Covariant Return Type

We briefly mentioned, in a previous video, that there's a clone method on the class Object, that all classes inherit from.

A simplified look at this declaration, for our purposes, is shown below.

```
protected Object clone() throws CloneNotSupportedException
```

And if you overrode this method, by using IntelliJ's code generation tools, it would generate this code in your class:

```
@Override
protected Object clone() throws CloneNotSupportedException {
    return super.clone();
}
```

Covariant Return Type

But in general, when you're cloning an instance, you're going to want to return an Object, that's the same type as the Object you are cloning.

Remember, we said all classes ultimately have Object as a base class, so every class can be said to be a covariant of Object.

Covariant Return Type

The clone method (generated by IntelliJ) override (from Object)

```
@Override
protected Object clone() throws CloneNotSupportedException {
    return super.clone();
}
```

The clone method overridden in a Person class.

This is a valid override of Object's clone method.

```
class Person {

    private String name;
    private String birthDate;

    public Person(String name, String birthDate) {
        this.name = name;
        this.birthDate = birthDate;
    }

    @Override
    public Person clone() {
        return new Person(name, birthDate);
    }
}
```